S-memV: Split Migration of Large-memory Virtual Machines in IaaS Clouds

Masato Suetake, Takahiro Kashiwagi, Hazuki Kizu, and Kenichi Kourai
Kyushu Institute of Technology
{*masato,kashiwagi,hazuki,kourai*}*@ksl.ci.kyutech.ac.jp*

***Abstract*—Recently, Infrastructure-as-a-Service clouds provide virtual machines (VMs) with a large amount of memory. Such large-memory VMs make VM migration difficult because it is costly to reserve large-memory hosts as the destination. Using virtual memory is a remedy for this problem, but virtual memory is *incompatible* with the memory access pattern in VM migration. Consequently, large performance degradation occurs during and after VM migration due to excessive paging. This paper proposes *split migration* of large-memory VMs with *S-memV*. Split migration migrates a VM to one main host and one or more sub-hosts. It divides the memory of a VM and transfers memory likely to be accessed to the main host. Since it transfers the rest of the memory directly to the sub-hosts, no paging occurs during VM migration. After split migration, remote paging is performed between the main host and the sub-hosts, but its frequency is lower thanks to memory splitting that is aware of remote paging. We have implemented S-memV in KVM and showed that the performance of split migration and application performance after VM migration were comparable to that of traditional VM migration with sufficient memory.**

I. Introduction

In Infrastructure-as-a-Service (IaaS) clouds, many virtual machines (VMs) are consolidated into a small number of hosts to reduce costs. Recently, as the needs to IaaS clouds are diversified, IaaS clouds also provide VMs with a large amount of memory. For example, Amazon EC2 provides the x1e.32xlarge instance type with 3.9 TB of memory. The M128ms instances in Microsoft Azure also have 3.8 TB of memory. Such large-memory VMs are required for big data analysis, e.g., using Apache Spark [1] and Facebook Presto [2], because big data can be analyzed efficiently by maintaining data in memory as much as possible. Fast in-memory databases such as SAP HANA [3] and Microsoft SQL Server [4] are other applications that use a large amount of memory.

There are two issues to migrate such large-memory VMs. One is the migration time because that time is basically proportional to the memory size of a migrated VM. This issue has been resolved by using fast interconnects such as 40 Gigabit Ethernet (GbE) [5] and parallelizing VM migration [6]. The other unresolved issue is the availability of the destination host. VM migration needs sufficient free memory at the destination host. However, it is costly to always reserve hosts with a large amount of free memory, even if possible in clouds. If large-memory VMs cannot be migrated, they have to be stopped during host maintenance and big data analysis is disrupted for a long time. In addition, the whole data in memory is lost and it takes much time to restore the lost data in memory by reading storage or redoing computation. This largely degrades performance for a long time after VMs are restarted.

To migrate a large-memory VM to a host with insufficient free memory, the virtual memory technology can be used. Virtual memory enables the system to run a VM with a larger amount of memory than physical memory by paging out part of the memory to storage. However, virtual memory is *incompatible* with the migration of a large-memory VM. Since VM migration transfers the entire memory of a VM to the destination host, memory that cannot be accommodated in the host is paged out. At this time, the page-outs are unconditionally done, regardless of the memory access pattern inside the VM. After VM migration, such paged-out memory is paged in again in a high probability. As such, using virtual memory causes excessive paging during and after VM migration and largely degrades the performance of VM migration and applications running in the migrated VM.

To solve this problem, this paper proposes *split migration* of large-memory VMs with *S-memV*. Split migration enables a large-memory VM to be migrated to multiple hosts by dividing its memory. In split migration, the destination of VM migration consists of one *main host* and one or more *sub-hosts*. Split migration transfers VM's core information such as CPU and device states to the main host. It also transfers memory likely to be accessed after VM migration to the main host as much as possible. In contrast, it transfers memory that cannot be accommodated in the main host *directly* to the sub-hosts. Therefore, paging does not occur at all during split migration. After split migration, the migrated VM runs at the main host and *remote paging* [7]–[11] is performed between the main host and the sub-hosts when the VM requires memory in the sub-hosts. Thanks to the awareness of locality of memory reference on memory splitting of a VM, the frequency of remote paging can be suppressed.

We have implemented S-memV in KVM and a memory server that manages part of the memory of a VM at a sub-host. We have developed a mechanism for obtaining the memory access history from the extended page tables (EPT) and implemented the least recently used (LRU) algorithm using temporal locality of reference. Also, we have extended the userfaultfd mechanism in Linux 4.3 for page-outs and

developed a remote paging system. According to our experiments, split migration could achieve much less migration time and downtime than VM migration with virtual memory. Rather, it was comparable to traditional VM migration with sufficient physical memory. These results came from no paging during VM migration. In addition, S-memV could suppress the degradation of application performance after split migration by predicting memory access.

The rest of this paper is organized as follows. Section II describes an issue in migrating VMs with a large amount of memory. Section III proposes split migration and Section IV describes its implementation. Section V shows experimental results of split migration with S-memV. Section VI describes related work and Section VII concludes this paper.

II. Migration of Large-memory VMs

VM migration enables a running VM to be moved to another host without stopping it. Using VM migration, administrators can maintain a host without service disruption after they migrate all the VMs running at that host. VM migration first creates a new VM at the destination host. In the first iteration, it copies the memory contents of a target VM running at the source host to the memory of the newly created VM via the network. In the following iterations, it re-transfers modified memory contents because the memory of the VM at the source host continues to be modified during the memory transfer. VM migration repeats the re-transfers and enters the last iteration when the amount of memory to be re-transferred is small enough. At this time, VM migration stops a VM at the source host and transfers the remaining modified memory and CPU and device states. At the destination host, it resumes the virtual devices using received device states and finally restarts the new VM.

Recently, VMs with a large amount of memory, e.g., 4 TB, are being widely used in IaaS clouds, but such large-memory VMs make VM migration difficult. This is because it is not cost-efficient to always reserve hosts with a large amount of free memory as the destination of VM migration. Larger-capacity memory modules are needed for large-memory hosts, but they are much more expensive than smaller-capacity ones. Also, inflexibility in managing large-memory hosts leads to higher management cost. If a large-memory host is used for running many small VMs, administrators have to first migrate these VMs to obtain necessary free memory. This is a time-consuming task and increases the time until the migration of a large-memory VM is completed.

When there is not sufficient free memory at the destination host for VM migration, the virtual memory technology is traditionally used. Using virtual memory, the system can run a VM with a larger amount of memory than physical memory. The memory pages that cannot be accommodated in physical memory are paged out to storage, as illustrated in Fig. 1. If a VM requires paged-out pages, the virtual memory

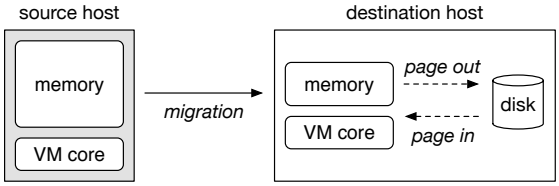


Figure 1: VM migration with virtual memory.

system pages in them to physical memory. Instead of the paged-in pages, it pages out unused pages to storage, usually, on the basis of the LRU algorithm. Frequent paging degrades VM performance largely, but several techniques have been proposed to suppress such performance degradation [12]. Using SSDs instead of HDDs as swap space also remedies this problem although even SSDs are still one or two orders of magnitude slower than memory.

However, virtual memory is *incompatible* with the migration of a large-memory VM. In the first iteration of VM migration, all the memory pages of a VM are transferred in order and pages that cannot be accommodated in physical memory are paged out to storage. Since all the pages are accessed only once in this iteration, the LRU algorithm usually used in paging is completely ineffective. This leads to a long migration time. Worse, since the page-outs are performed on the basis of the LRU algorithm, pages that have been transferred earlier are unconditionally paged out, regardless of the memory access pattern inside the VM. This can result in degrading the performance of virtual memory after VM migration.

In the following iterations, not only page-outs but also page-ins can occur frequently. When memory pages are re-transferred due to memory updates, those that are not resident in physical memory are first paged in from storage and then overwritten. At the same time, unused pages in physical memory are paged out. The number of such modified pages can become larger for a larger-memory VM because it takes a longer time to transfer all the memory pages in the first iteration. When VM migration is completed, frequently modified pages are likely to reside in physical memory. However, frequently accessed *read-only* pages can be paged out because VM migration does not re-transfer memory pages that are not modified. Therefore, paging is caused by accessing such read-only pages after VM migration.

In the last iteration, the occurrence of paging is critical because a VM is stopped during this iteration and performance degradation due to paging leads to a long downtime. Paging in this iteration can be caused by transferring modified memory pages, as in the previous iterations. If both the memory of a VM and the memory used for the virtual devices are managed by the same virtual memory system, as in KVM, resuming the virtual devices can cause paging. Since the virtual devices do not run yet at the destination

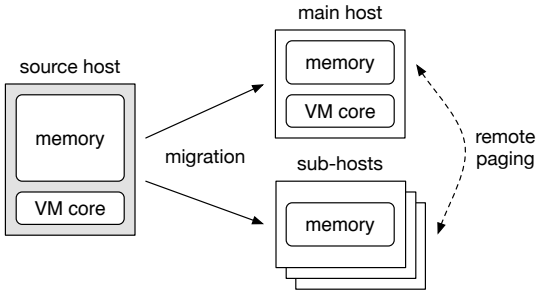


Figure 2: Split migration.

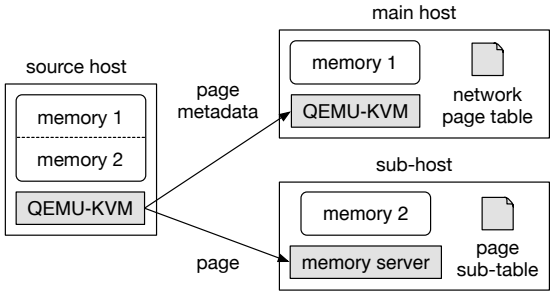


Figure 3: The system architecture in split migration.

host, most of their memory has been paged out. Therefore, many page-ins are necessary to write received device states to their memory and access the memory for re-initializing the virtual devices.

According to our experiments in Section V, it is shown that the migration time becomes 11.7 times longer and the downtime exceeds 30 seconds at worst when HDDs are used as swap space. Even using SSDs, the migration time is still 2.2 times longer and the downtime is near 4 seconds. Also, our experimental result shows that, even when SSDs are used, it takes 21 minutes to restore the performance of memcached [13] after VM migration.

III. Split Migration

In this paper, we propose *split migration* of large-memory VMs with *S-memV*. As illustrated in Fig. 2, a VM running in one host is migrated to multiple hosts in split migration. The destination hosts consist of one *main host* and one or more *sub-hosts*. The main host runs VM core such as virtual CPUs and devices with part of the memory of a VM, while sub-hosts manage the rest of the memory. Split migration divides a large amount of memory of a VM into smaller pieces and directly transfers them to these hosts. It transfers VM's core information and memory pages likely to be accessed to the main host. This enables the VM to access its memory without paging after VM migration. In contrast, split migration transfers memory pages that cannot be accommodated in the main host to sub-hosts.

After split migration, S-memV runs the migrated VM at the main host, performing *remote paging* [7]–[11] between the main host and the sub-hosts. Remote paging is a mechanism for paging in/out memory pages from/to the memory at other hosts via the network, instead of local storage. If the network is fast enough, remote paging is faster than paging with local storage. When a memory page accessed by the VM does not exist in the main host, S-memV pages in the requested memory page from one of the sub-hosts. To balance the amount of memory at the main host, it pages out a page unlikely to be accessed to the sub-host. For efficiency, S-memV pages in/out several pages including the target page at once.

In S-memV, remote paging is not caused at all *during* split migration. While a VM is being migrated, memory pages that cannot be accommodated in the main host are not paged out from the main host to the sub-hosts. Instead, they are directly transferred to the sub-hosts. Therefore, there is no wasteful network transfer between the main host and the sub-hosts at both the first memory transfer and the following re-transfers in VM migration. Also, the system load at the destination main host is not increased by remote paging. Without remote paging during VM migration, split migration can achieve less migration time and downtime.

On the other hand, S-memV requires remote paging *after* split migration, but the frequency is less than after traditional VM migration with virtual memory. This is because memory splitting in split migration is aware of remote paging performed after the migration. Split migration stores memory pages likely to be accessed in the memory of the main host so that remote paging occurs as infrequently as possible. To divide the memory of the VM in such a manner, S-memV monitors the memory access pattern of the VM. Using the obtained memory access history, it predicts future memory access on the basis of the LRU algorithm.

IV. Implementation

We have implemented S-memV in QEMU-KVM 2.4.1 and Linux 4.3. As illustrated in Fig. 3, the system consists of one source host, one destination main host, and one or more destination sub-hosts. The source host and the destination main host run QEMU-KVM in which S-memV is implemented and run VMs on top of it. Each destination sub-host runs a *memory server*, which manages part of the memory of VMs.

A. Migration Overview

To migrate a VM to multiple hosts, we have extended the migration mechanism of QEMU-KVM. In S-memV, QEMU-KVM at the source host (hereafter, source QEMU-KVM) connects to not only QEMU-KVM at the destination main host (hereafter, destination QEMU-KVM) but also the memory servers at the destination sub-hosts. The sub-hosts are chosen appropriately by a server that manages free

memory, the performance of CPU and memory, the system load, and network latency of all the hosts. Existing cloud management systems such as OpenStack already provide such servers for VM placement.

Next, the source QEMU-KVM splits the memory of a VM for the main host and chosen sub-hosts. This memory splitting is based on the LRU algorithm. S-memV assigns memory pages that are more likely to be accessed after VM migration to the main host in order, while it assigns the rest of the pages to the sub-hosts. S-memV performs memory splitting using the memory access history at the beginning of VM migration and does not consider memory access during VM migration. This is because the amount of transferred memory can increase if the destination host for each page changes during VM migration.

Once S-memV determines the destination host for each memory page, the source QEMU-KVM transfers memory data to either the destination QEMU-KVM or one of the memory servers. It can do that in parallel as proposed in [6]. At the re-transfer of modified pages, it transfers memory data to the same host. For a memory page accommodated in the main host, the source QEMU-KVM transfers a pair of the offset to a memory block and the data of the page to the destination QEMU-KVM as usual. In addition, it transfers the memory access history for the page so that the destination QEMU-KVM can continue to use the history for remote paging.

For a page accommodated in a sub-host, in contrast, the source QEMU-KVM transfers the physical memory address in the VM and the data of the page to the sub-host. The memory servers at sub-hosts maintain received pages using *page sub-tables*. In addition, the source QEMU-KVM also transfers the following information to the main host: the IP address of the destination sub-host and the offset to a memory block. The destination QEMU-KVM creates a *network page table* and maintains in which host each page is located to enable remote paging.

When VM migration is completed, the destination QEMU-KVM connects to the memory servers at the destination sub-hosts for remote paging. S-memV performs remote paging between the main host and one of the sub-hosts when it detects VM's access to the pages that are not located in the main host. If VM migration fails due to problems of the network or the destination hosts, it is aborted and the VM continues to run at the source host like traditional pre-copy migration. For fault tolerance after split migration, we could use a technique proposed in [10].

B. Memory Access History

To maintain the memory access history of a VM, S-memV keeps track of page access inside a VM. First, QEMU-KVM issues the extended ioctl system call to the modified KVM module in the Linux kernel to obtain information on memory access. At that time, it allocates an access bitmap whose bit corresponds to each memory page and passes it to the system call. Next, the KVM module traverses the extended page tables (EPT) for all the physical pages assigned to the target VM and examines the page table entries. Their access bit is set by a CPU with support for EPT A/D bit when the corresponding page is accessed. The KVM module records the value of the access bit to the passed bitmap. Finally, it resets the access bit so that CPUs can record new memory access in EPT for the next period.

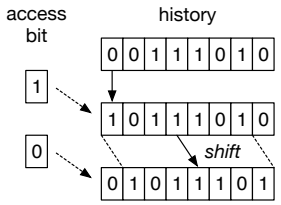


Figure 4: Updates in memory access history.

S-memV supports the aging algorithm as LRU approximation. For each page-out, S-memV finds a page that is least accessed for a certain period as a victim. Therefore, it requires a multi-bit memory access history for each page. In the current implementation, S-memV allocates eight bits to each page. S-memV periodically traverses EPT and accumulates the value of an obtained access bit in the most-significant bit of the memory access history for a while, as illustrated in Fig. 4. After a certain period, it shifts the memory access history to the right by one bit. This method enables S-memV to maintain a relatively long history with less bits and consider the latest memory access. Without this accumulation of access bits, S-memV could maintain the history only in eight seconds if it shifts the history every second. It could maintain a longer history if it obtains access bits every 10 seconds, but it could not consider memory access in the latest 10 seconds at worst.

C. LRU-based Memory Splitting

S-memV splits the memory of a VM on the basis of the memory access history on VM migration. This memory splitting is done in a granularity of a chunk, which consists of contiguous memory pages. The purpose of using a chunk is to achieve efficient remote paging. S-memV pages in a chunk including a faulting page located in a sub-host and pages out another chunk located in the main host.

To split the memory in the chunk granularity, S-memV needs to calculate the access history for each chunk, which is called the *chunk history*. The chunk history can be calculated by a bitwise OR of the 8-bit values of the memory access history for all the pages in a chunk, as illustrated in Fig. 5. For optimization, S-memV directly updates the chunk history whenever it obtains memory access information from EPT. Then, it assigns the pages in a chunk with a larger value to the main host in order.

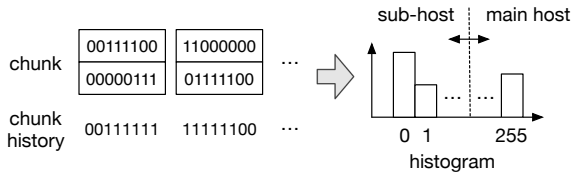


Figure 5: LRU-based memory splitting (chunk size of 2).

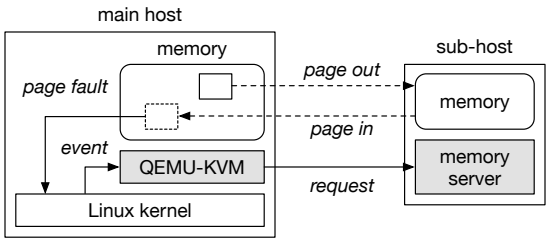


Figure 6: Remote paging using userfaultfd.

To achieve this assignment without sorting a large number of chunks, S-memV first creates a histogram about the 8-bit value of the chunk history. Next, it sums up the number of chunks in the histogram in descending order of history values until the sum exceeds the number of chunks that can be accommodated in the main host. The history value at that time is a threshold. S-memV assigns chunks with history values larger than the threshold to the main host. For chunks with the history value of the threshold, it assigns chunks to the main host as much as possible and the rest to the sub-hosts. It assigns chunks with smaller history values to the sub-hosts.

D. Memory Server

A memory server runs at a sub-host and manages part of the memory of a VM in a page granularity. For the memory management, it creates a *page sub-table*, which is a one-dimensional array whose index is a page frame number and whose value is the pointer to the data of the corresponding memory page. A memory server handles page-out and page-in requests. A page-out request consists of a physical memory address in a VM and the data of the corresponding memory page. When a memory server receives a page-out request, it allocates 4-KB memory, copies the received data to it, and adds it to the page sub-table. In contrast, a page-in request consists of only a physical memory address. When a memory server receives a page-in request, it searches the page sub-table and returns the data of the corresponding memory page. At the same time, it removes the data from the page sub-table and releases the memory for that data.

E. Remote Paging

To achieve remote paging for a VM at the main host, S-memV uses the userfaultfd mechanism, which was introduced in Linux 4.3. In KVM, the memory of a VM is allocated by anonymous memory mapping. When the destination QEMU-KVM receives data from the source host and writes it to the corresponding memory page, a physical memory page is assigned. For the other pages, physical memory is not assigned yet. To trap access to non-existent memory pages, the destination QEMU-KVM issues the userfaultfd system call at the end of split migration and registers all the memory pages of the migrated VM to userfaultfd.

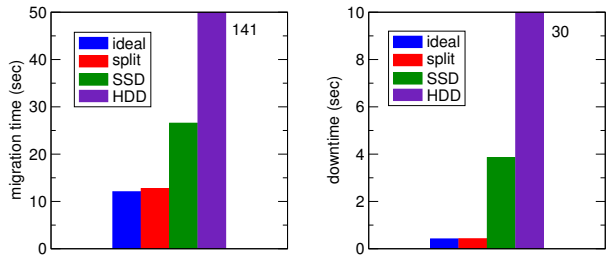
Fig. 6 illustrates how a memory page is paged in/out using userfaultfd. When a virtual CPU in the VM accesses a non-existent page, a page fault occurs and an event is notified to QEMU-KVM. QEMU-KVM translates the notified host memory address into the physical memory address used inside the VM. Then it sends page-in requests to the memory server at the sub-host that manages the corresponding memory pages. At this time, it first sends a request for the faulting page including the accessed memory address and then sends requests for the other pages in the chunk including the faulting page. This is because QEMU-KVM can handle the faulting page at first and continue to run the virtual CPU as early as possible. QEMU-KVM finds that sub-host by searching the *network page table*, which is a one-dimensional array whose index is a page frame number and whose value is the identifier of the host where the page is resident.

When QEMU-KVM receives the data of the corresponding page from the sub-host, it assigns physical memory to the page and writes the data using userfaultfd. At this time, the faulting virtual CPU in the VM is resumed. In addition, QEMU-KVM modifies the network page table so that the paged-in pages exist in the main host.

To balance the amount of memory, QEMU-KVM at the main host performs page-outs after page-ins. First, S-memV determines a paged-out chunk on the basis of the memory access history. Using the chunk history described in Section IV-C, S-memV finds a chunk with the smallest history value. Next, QEMU-KVM obtains the memory contents of the pages in the found chunk and removes the assignment of physical memory to those pages. To execute these two operations atomically with the VM running, we extended the userfaultfd mechanism so that it can clear the corresponding page table entries and return the page contents at the same time. Next, QEMU-KVM sends page-out requests for the pages in the chunk to the memory server at the sub-host where a chunk has been paged in. In addition, QEMU-KVM modifies the network page table so as to reflect the paged-out pages.

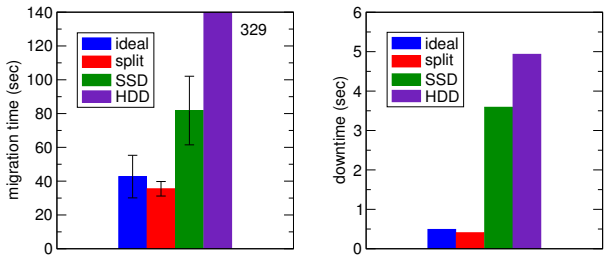
V. Experiments

To show the effectiveness of split migration, we measured migration performance and application performance after the



(a) Migration time (b) Downtime

Figure 7: Migration performance (idle).



(a) Migration time (b) Downtime

Figure 8: Migration performance (RapidWrite).

migration. For split migration, we used the chunk size of 256 pages, which achieved the best performance of remote paging. For comparison, we executed VM migration with virtual memory, which performed paging with local storage during and after the migration. We call VM migration using SSDs and HDDs as swap space *SSD-* and *HDD-assisted migration*, respectively. For the baseline, we used the traditional *ideal VM migration*, which migrated a VM to the destination host with sufficient free memory.

For the source host and the destination (main) host, we used two PCs with an Intel Xeon E3-1270v3 processor and 16 GB of DDR3 SDRAM. When executing split migration and VM migration with virtual memory, we adjusted free memory at the destination main host to the half of VM's memory. As swap space, we used 275 GB of Crucial MX300 SSD or 1 TB of SATA 3 HDD. For the destination sub-host, we used a PC with an Intel Xeon E3-1270v2 processor and 12 GB of DDR3 SDRAM. These PCs were connected with 10 Gigabit Ethernet (GbE). We ran Linux 4.3.0 and QEMU-KVM 2.4.1. For a VM, we assigned one virtual CPU and 12 GB of memory. This VM did not have so large memory, but that was suitable to analyze the performance in detail. In fact, that size of memory was necessary to complete HDD-assisted migration in a realistic time.

A. Performance of VM Migration

First, we migrated an idle VM without explicitly running applications inside it. The migration time is shown in Fig. 7(a). Compared with ideal migration, split migration increased the migration time only by 5%. For VM migration with virtual memory, in contrast, the migration time was 2.2 times longer even when SSD-assisted migration was executed. For HDD-assisted migration, the migration time was 11.7 times longer. This is because 6 GB of memory in the VM was paged out at the destination host during VM migration.

Fig. 7(b) shows the downtime of the migrated VM. For split migration, the downtime was only 7 ms longer than that in ideal migration. However, SSD- and HDD-assisted migration increased the downtime by 3.4 seconds and 30 seconds, respectively. These increases were caused by excessive paging. For an idle VM, the last iteration was started in the middle of the first iteration in KVM because of a small number of memory pages to be re-transferred. Therefore, many page-outs as in the first iteration occurred while a VM was stopped. In addition, many heap pages used by QEMU-KVM for the virtual devices were paged in.

Next, we migrated a memory-intensive VM, which ran a benchmark called RapidWrite. The benchmark allocated 6 GB of memory and repeatedly modified it. To avoid infinite memory re-transfers in VM migration, RapidWrite slept for five seconds after modifying 6 GB of memory.

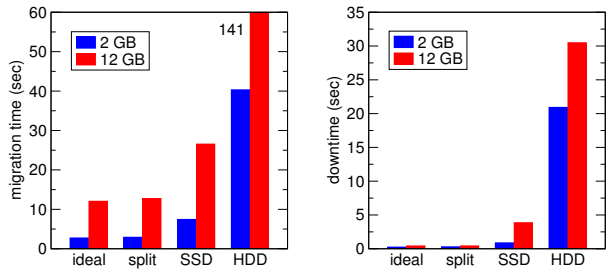
Fig. 8(a) shows the migration time when we migrated the memory-intensive VM. Compared with the migration of an idle VM, the migration time increased by 31 seconds even in ideal migration. For split migration, in contrast, the increase was only 23 seconds. This is probably due to the timing of executing the sleep of five seconds in RapidWrite. In fact, the variance of the migration time was quite large. For SSD- and HDD-assisted migration, the migration time was 55 and 188 seconds longer, respectively. This is because paging overhead during VM migration further increased the number of re-transferred pages.

On the other hand, the downtime was almost the same as that in an idle VM, as shown in Fig. 8(b), except for HDD-assisted migration. The downtime in HDD-assisted migration decreased by 26 seconds. When the memory-intensive VM is migrated, modified memory pages tend to be resident in physical memory at the destination host. As a result, paging due to memory re-transfers was less frequent in the last iteration.

B. Impact of VM Size and Network

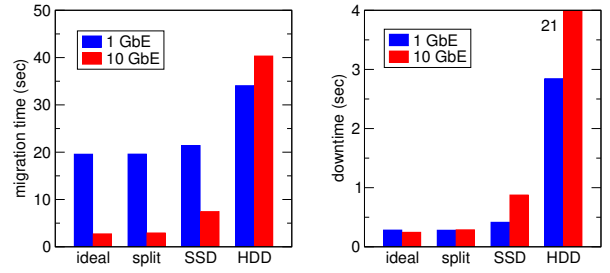
To investigate the impact of the memory size of a VM, we compared migration performance between two memory sizes of 2 GB and 12 GB. Fig. 9(a) shows the migration time of two idle VMs. Compared with the 2-GB VM, the migration time in the 12-GB VM was 3.5 to 4.4 times longer although the memory size was 6 times larger. The reason is that it takes a constant time to migrate a VM although the migration time is basically proportional to the memory size.

Fig. 9(b) shows the comparison of the downtime. For ideal and split migration, the downtime was only slightly increased in the 12-GB VM. For VM migration with virtual



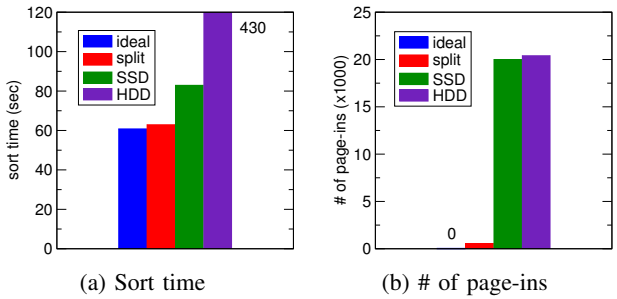
(a) Migration time (b) Downtime

Figure 9: The performance comparison of VM migration (2-GB VM vs. 12-GB VM).



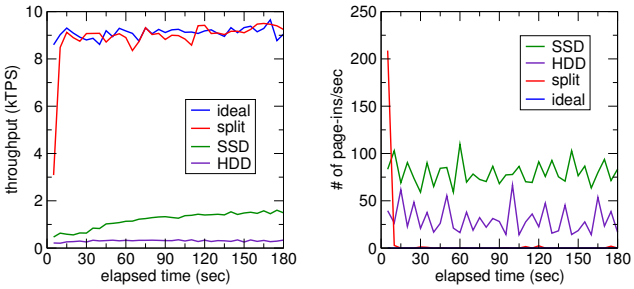
(a) Migration time (b) Downtime

Figure 10: The performance comparison of VM migration (1 GbE vs. 10 GbE).



(a) Sort time (b) # of page-ins

Figure 11: The sort performance after VM migration.



(a) Throughput (b) Page-in rate

Figure 12: The memcached performance after VM migration.

memory, in contrast, the downtime in the 12-GB VM increased largely. This is because the virtual memory system paged out approximately 700 MB of memory unused in several processes and assigned it to the VM. As a result, 83% of the VM's memory was resident in the 2-GB VM, while only 56% was resident in the 12-GB VM.

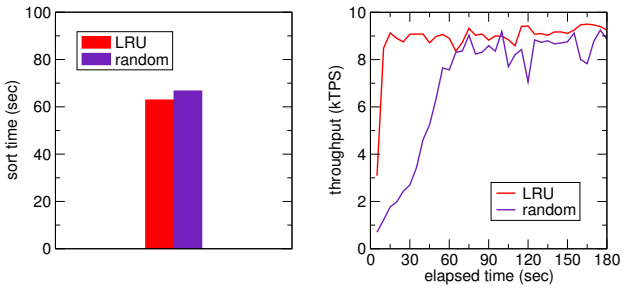
Next, we examined the impact of the network to migration performance. In this experiment, we used a 2-GB idle VM. Fig. 10(a) shows the migration time in 1 GbE and 10 GbE. Using 10 GbE, the migration time in split migration became 7 times shorter, as in ideal migration, although the network became 10 times faster. This is due to the constant overhead of VM migration. In contrast, SSD-assisted migration became only 2.9 times faster in 10 GbE. While the migration time was almost the same as ideal migration in 1 GbE, it was much longer in 10 GbE. This is because only slower 1 GbE could hide the performance degradation due to paging. Surprisingly, the migration time of HDD-assisted migration was shorter in 1 GbE than in 10 GbE. One reason is that the downtime is much longer in 10 GbE, which is discussed below.

Fig. 10(b) shows the downtime in 1 GbE and 10 GbE. For ideal and split migration, the downtime almost did not change. For SSD-assisted migration, however, the downtime was doubled in 10 GbE. Worse, HDD-assisted migration increased the downtime by 18 seconds. This is because much more pages had to be transferred in the last iteration when 10 GbE is used and more paging occurred. In KVM, VM migration enters the last iteration if the remaining memory can be transferred in 300 ms. Obviously, 10 GbE can transfer 10 times more data than 1 GbE.

C. Performance after VM Migration

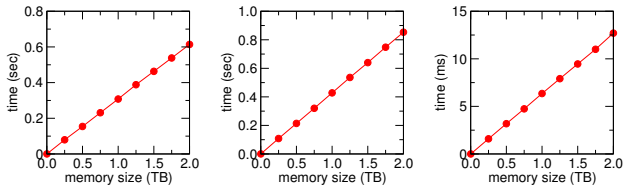
We measured the performance of GNU sort running in a VM after the migration. To obtain the memory access history, we first ran the sort command for 2 GB of data in 60 seconds. Then, we paused that process, migrated the VM, and resumed the process. Fig. 11 shows the time to complete the command and the number of page-ins during the command execution. Compared with ideal migration, performance degradation due to split migration was only 3.5%, whereas SSD-assisted migration degraded the performance by 36%. The reason of these differences is the number of page-ins. Split migration could suppress the number only to 2.8% of that in SSD-assisted migration. This means that split migration could transfer the memory accessed by the sort command to the main host successfully.

Next, we measured the performance of memcached [13] after VM migration. We assigned 5 GB of memory to memcached and accessed the data using the memaslap benchmark. We ran memaslap for 12 minutes before VM migration to obtain the memory access history, migrated the VM, and ran memaslap again. Fig. 12 shows the throughput and the number of page-ins every five seconds. After ideal migration, the throughput was always stable. For split



(a) Sort (b) memcached

Figure 13: The effectiveness of using the LRU algorithm.



(a) History collection (b) Memory splitting (c) Page-out

Figure 14: The overhead of using LRU.

migration, the performance was degraded by 66% just after the migration due to remote paging, but that was recovered only in 5 seconds. The network bandwidth consumed for this performance recovery was 3.4 Gbps, which was not so large in 10 GbE. The stable throughput was only 0.6% lower than that after ideal migration. On the other hand, performance degradation after VM migration with virtual memory was critical because of thrashing. For SSD-assisted migration, it took 21 minutes to restore the same performance as before the migration.

When the working set size exceeded the memory size of the main host, remote paging largely affected the performance of memcached after split migration. For example, the throughput was degraded by 69% for memcached with 6 GB of memory assigned. However, remote paging with RDMA [9], [11] could suppress such performance degradation, e.g., up to 27% in memcached.

D. Effectiveness of Using LRU

To show the effectiveness of LRU approximation, we randomly performed memory splitting and page-outs in remote paging and compared application performance after VM migration. In this experiment, we executed GNU sort and memcached as in Section V-C. Fig. 13 shows the execution time of sort and the throughput of memcached after VM migration. For GNU sort, the execution time in the LRU algorithm was 5.7% shorter than that in the random algorithm. For memcached, unlike the LRU algorithms, it took 60 seconds to recover the throughput in the random algorithm. Even after performance recovery, the throughput was 6.7% lower. These show that using LRU algorithms for memory splitting and page-outs is effective.

E. Overhead of Using LRU

We examined the overhead of using LRU in S-memV. In this experiment, we executed three mechanisms using LRU independently of the Linux kernel and QEMU-KVM so that we could perform the measurement regardless of the size of physical memory. We changed the memory size from 1 GB to 2 TB. We configured access bits in EPT and memory access history randomly and split the memory into two equally.

Fig. 14(a) shows the time for the collection of memory access history. The collection time was proportional to the memory size and 0.6 seconds for 2-TB memory. Although this overhead is not small, parallel collection using multiple CPUs could reduce the time. Fig. 14(b) shows the time for memory splitting. For 2-TB memory, it took 0.8 seconds to split the memory of a VM, but this is negligible, compared with a long migration time. Fig. 14(c) shows the time for page-out decision. The decision time was 12.7 ms for 2-TB memory, but this overhead would not largely affect the performance of remote paging because page-outs are less critical than page-ins.

VI. Related Work

Post-copy VM migration [14] is a special case of split migration in that it runs a VM using two hosts for a while. While the migrated VM is running at the destination host, the source host transfers the memory of the VM on demand or in the background like page-ins from a sub-host. However, since this situation is transient, the VM finally runs at only one destination host after VM migration is completed. In addition, difficult page-out decision is not necessary.

Scatter-Gather live migration [15] uses multiple intermediate hosts between the source and destination hosts. It pushes the memory of a VM to intermediate hosts as fast as possible and the destination host obtains the memory from these hosts. This is similar to split migration in that the source host transfers the memory to multiple hosts and that the destination host pages in that from remote hosts. However, Scatter-Gather migration does not need to predict memory access of VMs to reduce remote paging. Since it finally migrates a VM to one destination host, it does not need page-outs, resulting in simpler paging.

Unlike S-memV, MemX [10] always runs a VM using the memory of multiple hosts. In the MemX-VM mode, the guest operating system in a VM provides a block device to access the memory at the other hosts. In the MemX-DD mode, Dom0 in Xen provides such a block device. In the MemX-VMM mode, MemX provides a VM with the memory extension to access the memory at the other hosts transparently. When a VM accesses a memory page that does

not exist at the host, MemX obtains the corresponding page at another host and assigns it to the VM. This mechanism is similar to remote paging in S-memV, but only the MemX-VM mode allows VM migration.

vNUMA [16] enables running one large VM with not only the memory but also CPUs of multiple hosts. The VM can transparently access the memory of all the hosts using distributed shared memory. While S-memV uses only one host for running a VM and the other hosts for providing memory, these systems use all the hosts for running a VM. Therefore, the overhead for the cooperation between multiple hosts is much larger.

The optimization for restoring checkpointed VMs can be applied for performance improvement after split migration. Working set restore [17] prefetches the working set of the VM's memory from a disk. For working-set estimation, it scans access bits in the page tables of the guest operating system. Halite [18] groups VM's memory pages likely to be accessed together into locality blocks upon checkpointing. To predict access locality of memory pages, Halite uses the locality in virtual address spaces of the guest operating system.

VII. Conclusion

This paper proposed split migration with S-memV, which divides the memory of a large-memory VM into small pieces and directly transfers them to multiple hosts. Split migration transfers the memory likely to be accessed to the destination main host and the remaining memory to the sub-hosts. Unlike VM migration with virtual memory, paging does not occur at all during VM migration. After split migration, the VM runs at the main host and S-memV performs remote paging between the main host and sub-hosts when necessary. Thanks to LRU-based memory splitting, the frequency of remote paging is much lower. We have implemented S-memV in KVM and confirmed that the performance of split migration was comparable to that of ideal VM migration. In addition, S-memV could suppress the degradation of application performance after split migration.

One of our future work is to evaluate split migration in various configurations. We are interested in the migration of larger-memory VMs, e.g., with 4 TB of memory, using faster networks such as 100 GbE with RDMA. We need to also compare the performance with SSD-assisted migration using faster NVMe SSDs. In addition, we are planning to develop the migration of a VM running across multiple hosts after split migration. For example, a split-memory VM should be merged into one host again if possible. Fault tolerance after split migration is another challenge. We can apply the RAID-like recovery mechanism that has been proposed in [10]. Another direction is to apply the mechanisms of split migration to VM migration with virtual memory, where swap space is considered a sub-host.

References

[1] Apache Software Foundation, "Apache Spark – Lightning-Fast Cluster Computing," http://spark.apache.org/.

[2] Facebook, Inc., "Presto: Distributed SQL Query Engine for Big Data," https://prestodb.io/.

[3] SAP SE, "SAP HANA," https://hana.sap.com/.

[4] Microsoft Corporation, "SQL Server 2014," https://www.microsoft.com/en/server-cloud/products/sql-server/.

[5] Mellanox Technologies, "Accelerating Virtual Machine Migration over vSphere vMotion and Mellanox End-to-End 40GbE Interconnect Solutions," http://www.mellanox.com/related-docs/solutions/SB_Accelerating_Virtual_Machine_Migration.pdf, 2016.

[6] X. Song, J. Shi, R. Liu, J. Yang, and H. Chen, "Parallelizing Live Migration of Virtual Machines," in *Proc. Int. Conf. Virtual Execution Environments*, 2013, pp. 85–96.

[7] D. Comer and J. Griffioen, "A New Design for Distributed Systems: The Remote Memory Model," in *Proc. Summer 1990 USENIX Conf.*, 1990, pp. 127–135.

[8] M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath, "Implementing Global Memory Management in a Workstation Cluster," in *Proc. Symp. Operating Systems Principles*, 1995, pp. 201–212.

[9] S. Liang, R. Noronha, and D. K. Panda, "Swapping to Remote Memory over InfiniBand: An Approach Using a High Performance Network Block Device," in *2005 IEEE Int. Conf. Cluster Computing*, 2005.

[10] U. Deshpande, B. Wang, S. Haque, M. Hines, and K. Gopalan, "MemX: Virtualization of Cluster-Wide Memory," in *Proc. Int. Conf. Parallel Processing*, 2010, pp. 663–672.

[11] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. Shin, "Efficient Memory Disaggregation with Infiniswap," in *Symp. Networked Systems Design and Implementation*, 2017, pp. 649–667.

[12] N. Amit, D. Tsafrir, and A. Schuster, "VSwapper: A Memory Swapper for Virtualized Environments," in *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 2014, pp. 349–366.

[13] B. Fitzpatrick, "memcached – A Distributed Memory Object Caching System," http://memcached.org/.

[14] M. R. Hines and K. Gopalan, "Post-Copy Based Live Virtual Machine Migration Using Adaptive Pre-Paging and Dynamic Self-Ballooning," in *Proc. Int. Conf. Virtual Execution Environments*, 2009, pp. 51–60.

[15] U. Deshpande, Y. You, D. Chan, N. Bila, and K. Gopalan, "Fast Server Deprovisioning through Scatter-Gather Live Migration of Virtual Machines," in *Proc. Int. Conf. Cloud Computing*, 2014, pp. 376–383.

[16] M. Chapman and G. Heiser, "vNUMA: A Virtual Shared-Memory-Multi Processor," in *Proc. USENIX Annual Technical Conf.*, 2009.

[17] I. Zhang, A. Garthwaite, Y. Baskakov, and K. Barr, "Fast Restore of Checkpointed Memory Using Working Set Estimation," in *Proc. Int. Conf. Virtual Execution Environments*, 2011, pp. 87–98.

[18] I. Zhang, T. Denniston, Y. Baskakov, and A. Garthwaite, "Optimizing VM Checkpointing for Restore Performance in VMware ESXi," in *Proc. USENIX Annual Technical Conf.*, 2013, pp. 1–12.